

Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework

Louis-Noël Pouchet
The Ohio State University
pouchet@cse.ohio-state.edu

Uday Bondhugula
IBM T.J. Watson Research Center
ubondhug@us.ibm.com

Cédric Bastoul
Paris-Sud 11 University
cedric.bastoul@u-psud.fr

Albert Cohen
INRIA Saclay – Île-de-France
albert.cohen@inria.fr

J. Ramanujam
Louisiana State University
jxr@ece.lsu.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

Abstract—Today’s multi-core era places significant demands on an optimizing compiler, which must parallelize programs, exploit memory hierarchy, and leverage the ever-increasing SIMD capabilities of modern processors. Existing model-based heuristics for performance optimization used in compilers are limited in their ability to identify profitable parallelism/locality trade-offs and usually lead to sub-optimal performance.

To address this problem, we distinguish optimizations for which effective model-based heuristics and profitability estimates exist, from optimizations that require empirical search to achieve good performance in a portable fashion. We have developed a completely automatic framework in which we focus the empirical search on the set of valid possibilities to perform fusion/code motion, and rely on model-based mechanisms to perform tiling, vectorization and parallelization on the transformed program. We demonstrate the effectiveness of this approach in terms of strong performance improvements on a single target as well as performance portability across different target architectures.

I. INTRODUCTION

Realizing the high levels of potential performance on current machines is a very difficult task. One of several approaches to addressing this challenge is to develop compiler transformations aimed in particular at loops. This requires a compiler to be able to apply complex sequences of loop transformations and effectively model the effect of hardware resources and the complex ways in which they interact. Model-driven optimization heuristics used in current research and production compilers apply a restricted subset of the possible program transformations, thereby limiting their effectiveness. The problem becomes further complicated when one aims for performance portability over a broad range of architectures.

The polyhedral representation of programs enables the expression of arbitrarily complex sequences of loop transformations. But the downside to this expressiveness is the extreme difficulty in selecting a good optimization strategy combining the most important loop transformations, including loop tiling, fusion, distribution, interchange, skewing, permutation and shifting [1], [2]. It is also hard to analytically capture interacting effects of different hardware resources taking into account downstream optimization passes.

The state-of-the-art in tiling and parallelization in the polyhedral model [3] relies on an analytical approach. Unfortu-

nately, a purely analytical approach is not sufficient since it is difficult to adequately account for several high-impact factors such as cache conflicts, memory bandwidth and vectorization. It is important to adapt the optimization strategy to a target architecture; in addition, it is important to achieve portable performance, requiring understanding and management of the interplay between scalability, locality and synchronization overhead on different target machines.

To address these challenges, we have designed a combined iterative and model-driven scheme for optimization and parallelization. It relies on an iterative, feedback-directed exploration of loop structure choices, i.e., loop fusion/distribution choices. In turn, each fusion/distribution choice drives model-based algorithms for many other loop transformations including loop tiling and vectorization. Portability of performance is achieved thanks to iteratively testing different program versions. In practice, our method found the best version on all benchmarks we tested it on. We obtained improvements ranging from $1\times$ to $8.5\times$ over version produced by reference parallelizing compilers using model-based heuristics, and validated the portability of our approach considering two modern multi-core architectures (Intel Dunnington and AMD Shanghai) as well as a low-power embedded Intel Atom processor. The memory hierarchies and interconnects of these architectures are very different (e.g., front-side bus vs. point-to-point links). We chose these architectures in order to demonstrate the potential of our approach to achieve performance portability and to gauge the sensitivity to the underlying memory hierarchy.

The rest of this paper is organized as follows. Section II describes the motivation and presents the problem statement. Section III recalls the fundamental concepts in polyhedral models of compilation. Section IV addresses the construction, pruning and traversal of the search space. Section V presents the model-based optimization algorithms used in our combined strategy. Section VI evaluates this technique experimentally. Section VII discusses related work, before the conclusion in Section VIII.

II. PROBLEM STATEMENT

Achieving a high level of performance for a computational kernel on a modern multicore architecture requires effective mapping of the kernel onto the hardware resources that exploits

- thread-level parallelism;
- the memory hierarchy, including prefetch units, different cache levels, memory buses and interconnect; and
- all available computational units, including SIMD units.

Because of the very complex interplay among these, translating specific properties on the input code (e.g., the number of parallel loop iterations) into actual performance metrics is a significant challenge. Considering several high-level program transformations that expose the same amount of thread-level parallelism, determining which of these results in highest performance is beyond the reach of optimizing compilers. This is due at least in part to the combined effectiveness of hardware resources such as the memory hierarchy and the vector units, which can be significantly different on different targets.

To maximize performance, one must carefully explore the trade-off between the different levels of parallelism and the usage of the components of local memory. Maximizing data locality may be detrimental to inner-loop-parallelism, and may counter the effects of an efficient, well-aligned vectorized inner loop. On the other hand, focusing the optimization towards the most efficient vectorization may require excessive loop distribution, resulting in poor data reuse and thus may adversely affect performance.

Key program transformations for a loop nest optimizer are:

- thread-level parallelism extraction, to expose coarse-grain parallelism and benefit from the different hardware threads available;
- loop tiling, to improve the locality of computation and reduce the number of cache misses;
- SIMD-level parallelism extraction, by forming innermost loops which can be effectively vectorized;

Most previous efforts have taken a multi-stage, decoupled approach to choose the optimizations: they first identify a transformation to expose thread-level parallelism, then try to apply tiling on the resulting code, and finally try to vectorize the output [4], [5]. Recent work by Bondhugula et al. [3] integrated the extraction of parallelism and the identification of tileable loop nests, but did not take the memory hierarchy into account when selecting which loops to fuse and to parallelize, and did not consider the effect of these transformations on SIMD parallelism. As a result of this approach, reuse is maximized and parallel tiled code is generated, but this can lead to sub-optimal performance since maximizing locality can disable the vectorization of inner-loops and increase cache interference conflicts.

This leads us to the key observation that *loop fusion and distribution drive the success of subsequent optimizations, such as vectorization, tiling and array contraction*. The number of vectorizable or tileable loops is related to the set of statements that are fused under a common loop. The loop structure can

be seen as resulting from a *partitioning of the program*, where the statements in the same class of the partition all share at least one common outer-loop. We now study the impact on performance of different ways to partition the program, highlighting the need for target-specific tuning of the program partition. Let us consider the example 2mm, a kernel involving a sequence of two matrix multiplications $D = A.B.C$, as shown in Figure 1.

```

for (i1 = 0; i1 < N; ++i1)
  for (j1 = 0; j1 < N; ++j1) {
R:   tmp[i1][j1] = 0;
    for (k1 = 0; k1 < N; ++k1)
S:     tmp[i1][j1] += A[i1][k1] * B[k1][j1];
  }
  for (i2 = 0; i2 < N; ++i2)
    for (j2 = 0; j2 < N; ++j2) {
T:     D[i2][j2] = 0;
      for (k2 = 0; k2 < N; ++k2)
U:       D[i2][j2] += tmp[i2][k2] * C[k2][j2];
    }
  }

```

Fig. 1. Original code: $tmp = A.B$, $D = tmp.C$

We now observe that there exist 12 different valid partitions for this program, i.e., all possible ways to combine the textual statements R , S , T , U provided R is before T , T is before U , and S is before U . Figure 2 presents the result of a purely model-driven approach geared towards (a) minimizing communication and maximizing the data locality for the full program, and (b) exposing thread-parallelism and tileable loops [3]. This corresponds to partitioning the program such that all statements are in the same class: $p_{maxfuse} = \{\{R, S, T, U\}\}$. A complex sequence of loop transformations that includes skewing is needed to implement this partitioning. We note that for this example and the following, to enhance readability we omit the pragmas for OpenMP parallelization and vectorization as well as the tiling loops.

```

parfor (c0 = 0; c0 < N; c0++) {
  for (c1 = 0; c1 < N; c1++) {
R:   tmp[c0][c1] = 0;
T:   D[c0][c1] = 0;
    for (c6 = 0; c6 < N; c6++)
S:     tmp[c0][c1] += A[c0][c6] * B[c6][c1];
    parfor (c6 = 0; c6 <= c1; c6++)
U:       D[c0][c6] += tmp[c0][c1-c6] * C[c1-c6][c6];
  }
  for (c1 = N; c1 < 2*N - 1; c1++)
    parfor (c6 = c1-N+1; c6 < N; c6++)
U:       D[c0][c6] += tmp[c0][c1-c6] * C[c1-c6][c6];
  }
}

```

Fig. 2. Minimal communication, maximal fusion ($p_{maxfuse}$)

On a 4-socket Intel Xeon hexa-core 7450 server (24 cores, Dunnington microarchitecture), this transformation leads to a $2.4\times$ speedup over Intel's compiler ICC 11.1 with automatic parallelization enabled, with $N=1024$ using double-precision arithmetic. However, using the partitioning $p_{xeon} = \{\{R\}, \{S\}, \{T\}, \{U\}\}$, shown in Figure 3, leads to a $3.9\times$ speedup over the original code, and is the best partitioning for this machine. This partitioning no longer minimizes communication and synchronization, but on the other hand exposes inner parallel loops for all statements.

```

parfor (i1 = 0; i1 < N; ++i1)
  parfor (j1 = 0; j1 < N; ++j1)
R:   tmp[i1][j1] = 0;
  parfor (i1 = 0; i1 < N; ++i1)
    for (k1 = 0; k1 < N; ++k1)
      parfor (j1 = 0; j1 < N; ++j1)
S:   tmp[i1][j1] += A[i1][k1] * B[k1][j1];
  parfor (i2 = 0; i2 < N; ++i2)
    parfor (j2 = 0; j2 < N; ++j2)
T:   D[i2][j2] = 0;
  parfor (i2 = 0; i2 < N; ++i2)
    for (k2 = 0; k2 < N; ++k2)
      parfor (j2 = 0; j2 < N; ++j2)
U:   D[i2][j2] += tmp[i2][k2] * C[k2][j2];

```

Fig. 3. Best loop structure for Intel Xeon 7450 with ICC (p_{xeon})

For a 4-socket AMD Opteron quad-core 8380 server (16 cores, Shanghai microarchitecture), the best partitioning is $p_{opteron} = \{\{R\}, \{T, S\}, \{U\}\}$, and is shown in Figure 4.

```

parfor (c1 = 0; c1 < N; c1++)
  parfor (c2 = 0; c2 < N; c2++)
R:   C[c1][c2] = 0;
  parfor (c1 = 0; c1 < N; c1++)
    for (c3 = 0; c3 < N; c3++) {
T:   E[c1][c3] = 0;
      parfor (c2 = 0; c2 < N; c2++)
S:   C[c1][c2] += A[c1][c3] * B[c3][c2];
    }
  parfor (c1 = 0; c1 < N; c1++)
    for (c3 = 0; c3 < N; c3++)
      parfor (c2 = 0; c2 < N; c2++)
U:   E[c1][c2] += C[c1][c3] * D[c3][c2];

```

Fig. 4. Best loop structure for AMD Opteron with ICC ($p_{opteron}$)

Using the partitioning $p_{opteron}$ on the Intel Xeon performs 20% slower than the partitioning in Figure 3, while using p_{xeon} on the AMD Opteron performs 25% slower than $p_{opteron}$. To further study the impact of the partitioning on performance, we performed a similar analysis on a low-power Intel Atom 230 processor (single-core, 2 hardware threads, Diamondville microarchitecture). For this case, the best found partitioning is $p_{atom} = \{\{R, T\}, \{S, U\}\}$, for a 10% improvement over $p_{opteron}$ and p_{xeon} , and a $3.5\times$ improvement over $p_{maxfuse}$. Data locality and vectorization dominates on the Atom, leading to a completely different optimal partitioning.

We summarize these results in Figure 5 where we report the performance improvement over the original code (Improv.) for these three architectures, and the performance improvement over the partitioning with the best average on the three machines (Variability).

	Xeon	Opteron	Atom
Improv.	3.6×	8.3×	31.3×
Variability	20%	11%	14%

Fig. 5. Performance improvement and variability

The main performance difference between these program versions is not due to the exploitation of any single hardware resource; rather, it is because of the complex interaction between parallelization, vectorization and data cache utilization. From the point of view of high-level program transformations, it is the loop nest structure derived from program partitioning

that has the highest impact on performance.

Our approach to program optimization decouples the search of the *program structure* from the application of other performance-enhancing transformations, for example, for locality improvement and vectorization. The first step of our optimization process is to compute all valid partitions of the program statements, such that a class of this partition corresponds to a set of *fusible statements*: those statements that share at least one common loop in the target code. In the second step, for each valid partitioning, we apply model-driven optimizations individually on each class of the partition in a systematic fashion. These optimizations may lead to complex compositions of affine loop transformations (skewing, interchange, multi-level distribution, fusion, peeling and shifting). Part of the sequence is computed in the polyhedral abstraction to create outer loop(s) parallel and permutable when possible, optimizing for data locality; details are provided in Section V-A. Then, it is further modified in order to expose parallel inner-loops with a minimal reuse distance to enable efficient vectorization; this is covered in Section V-B.

It is very hard to predict the profitability of a program partitioning, due to the combinatorial nature of the problem and due to the very complex and apparently chaotic interaction among transformations resulting from the selection of a given partition. This interplay is machine-specific, and to find a profitable partition for a program we will thus resort to an iterative, feedback-directed search. On the other hand, profitability of optimizations such as tiling or vectorization are easier to assess, typically because they are generally beneficial if they do not destroy some other properties of the code, such as thread-level parallelism or data locality. We will thus rely on performance models to drive these choices.

III. PROGRAM OPTIMIZATION

Most internal representations used in compilers match the inductive semantics of imperative programs (syntax tree, call tree, control-flow graph, SSA etc.). In such reduced representations of the dynamic execution trace, a statement of a high-level program occurs only once, even if it is executed many times (e.g., when enclosed within a loop). This is not convenient for optimizations that need a granularity of representation reflecting dynamic *statement instances*. For example, transformations like loop interchange, fusion or tiling operate on the execution order of statement instances [6]. In addition, a rich algebraic structure is required when building complex compositions of such transformations [1], enabling efficient heuristics for search space construction and traversal [2].

A. Polyhedral Model

The *polyhedral model* is a flexible and expressive representation for loop nests with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [7], [1], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop

iterators and global variables (constants that are unknown at compile-time). Relaxation of these constraints based on affine over-approximations have been proposed recently [8]. While our optimization scheme is compatible with the recent proposal [8], we limit the presentation in this paper to describing the representation and optimization of standard SCoPs.

1) *Step 1: Representing programs*: Program optimization in a polyhedral model is a three stage process. First, the program is analyzed to extract its polyhedral representation, including dependence information and access pattern. For all textual statements in the program, for example R in Figure 1, the set of its dynamic instances is captured with a set of affine inequalities. When the statement is enclosed by loop(s), all iterations of the loop(s) are captured in the iteration domain of the statement. Considering the 2mm kernel in Figure 1, the iteration domain of R is:

$$\mathcal{D}_R = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i < N \wedge 0 \leq j < N\}.$$

The iteration domain \mathcal{D}_R contains only integer vectors (or, integer points if only one loop encloses the statement R). The *iteration vector* \vec{x}_R is the vector of the surrounding loop iterators; for R it is (i, j) and takes values in \mathcal{D}_R . Each vector in \mathcal{D}_R corresponds a specific set of values taken by the surrounding loop iterators (starting from the outermost to the innermost enclosing loop iterator) when R is executed.

The sets of statement instances between which there is a producer-consumer relationship are modeled as equalities and inequalities in a *dependence polyhedron*. This is defined at the granularity of the array cell. If two instances \vec{x}_R and \vec{x}_S refer to the same array cell and one of these references is a write, then they are said to be in dependence. Hence, to respect the program semantics, the transformed program must execute \vec{x}_R before \vec{x}_S . Given two statements R and S , a dependence polyhedron, written as $\mathcal{D}_{R,S}$ contains all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$.

Multiple dependence polyhedra may be required to capture all dependent instances, at least one for each pair of array references accessing the same array cell (scalars being a particular case of array). Hence it is possible to have several dependence polyhedra per pair of textual statements, as some may contain multiple array references.

2) *Step 2: Representing optimizations*: The second step in polyhedral program optimization is to compute a transformation for the program. Such a transformation captures in a single step what may typically correspond to a sequence of several tens of textbook loop transformations [1]. It takes the form of a carefully crafted affine multidimensional schedule, together with (optional) iteration domain or array subscript transformations. In this work, a given loop nest optimization is defined by a multidimensional affine schedule. Given a statement S , we use an affine form on the d enclosing loop iterators \vec{x}_S and p program parameters \vec{n} . It is written

$$\Theta^S(\vec{x}_S) = \begin{pmatrix} \theta_{1,1} & \cdots & \theta_{1,d+p+1} \\ \vdots & & \vdots \\ \theta_{m,1} & \cdots & \theta_{m,d+p+1} \end{pmatrix} \cdot \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$

where Θ^S is a *matrix of non-negative integer constants*. A schedule is a function which associates a logical execution date (a timestamp) to each instances of a given statement. In the case of multidimensional schedules ($m > 1$ in the above), this timestamp is a vector. In the target program, statement instances will be executed according to the increasing lexicographic order of their timestamp. To construct a full program optimization, we build a collection of schedules $\Theta = \{\Theta^{S^1}, \dots, \Theta^{S^n}\}$ such that for all dependent instances the producer instance is scheduled before the consumer one. Note that every static control program has a multidimensional affine schedule [9], and that any loop transformation can be represented in the polyhedral representation [6].

We limit the coefficients of Θ to be non-negative, which is a somewhat narrower definition of Θ used in general polyhedral theory. The motivation for this comes from the algorithm we use to select schedules for tiling: using $\theta_{i,j} \in \mathbb{Z}$ (the set of all integers) breaks the convexity of the scheduling problem and requires a combinatorial search [10]. On the other hand, forcing $\theta_{i,j} \in \mathbb{N}$, we significantly increase the scalability of the scheduling computation process. The drawback is losing loop reversal and some combinations of loop skewing from the search space, however extensive experiments indicate that is is not an issue for most programs.

Multidimensional polyhedral tiling is applied by modifying the iteration domain of the statements to be tiled, in conjunction with further modifications of Θ [1], [3].

3) *Step 3: Applying optimizations*: The last step is to generate a transformed program according to the optimization we have previously computed. Syntactically correct transformed code is generated back from the polyhedral representation on which the optimization has been applied. We use the CLOOG, a state-of-the-art code generator [11] to perform this task.

B. Combined Iterative and Model-Driven Optimization

In this work, we have organized the task of optimization and automatic parallelization of a loop nest as an iterative, feedback-directed search. Each iteration of the search is further decomposed into two stages:

- 1) choosing a partition of the program statements, such that statements inside a given class can share at least one common loop in the generated code;
- 2) on each class of this partition, applying a series of model-driven affine loop transformations: (a) a tiling-based optimization and parallelization algorithm; (b) a vectorization-based algorithm.

Our approach differs significantly from previous work using iterative compilation to search for an affine multidimensional schedule [2] in that we do not require an empirical search of the entire set of sequences of transformations. Instead, we limit the search only to the part for which no robust performance models have been derived, but rely on well-understood cost models for the other transformations. We aim for a substantial reduction of the search space size while still preserving the ability to explore the most important set of candidate transformations.

IV. CONSTRUCTING VALID PARTITIONS

Grouping statements such that those in a given class share at least one common enclosing loop is a way to abstract the essence of loop fusion; this general idea also enables the modeling of loop distribution and code motion. If some statements in a given class were not fusible, then this partitioning would be equivalent to the one where the statements are distributed: this is a case of duplication in the search space. Our approach to guarantee that we find the most effective partitioning is to exhaustively evaluate all of them. It is important to remove duplicates in the search space to minimize the time for empirical search.

On the other hand, retaining expressiveness is a major objective, as we aim to build a search space of *all valid* (that is, semantics-preserving) partitions of the program. To achieve this goal, we leverage the expressiveness of the polyhedral representation and its ability to compute arbitrary enabling transformations (e.g., permutation, skewing, etc.) for fusion. We first provide a practical definition for fusion of statements in the polyhedral model in Section IV-A, before discussing the construction of the search space of all valid partitions in Section IV-B.

A. Loop fusion and fusibility of statements

The commonly used (syntactical) approach to loop fusion requires matching bounds for the loops to be fused; otherwise prolog and epilog code need to be explicitly generated for the remaining loop iterations. This problem becomes difficult when considering imperfectly nested loops with parametric bounds. However, using a polyhedral abstraction, the process of generating prolog/epilog code for arbitrary affine loops is handled seamlessly by the code generation step. The only task is to provide a schedule for the program that corresponds to fusing some statement instances inside a common loop. In this representation, loop fusion can be seen as the negation of loop distribution. Two statements R and S are distributed if all instances of R are scheduled to execute before the first (or after the last) instance of S . For any other case, there is an overlap and some instances of R and S are interleaved. Given a loop level, we distinguish the *statement interleaving* that describes R being executed fully before or after S (no common loop), from the *fine-grain interleaving* of statement instances where R and S share a common loop.

In this paper, we use a stricter definition of fusion in the polyhedral representation. The cases where only a few instances are fused and most of the loop iterations end up in the prolog/epilog parts offer little interest in our framework. In such cases, the core of the computation remains in the distributed loops, and since we aim at exploring all distinct fusion/distribution choices, it is likely that this variant would offer little difference with the case where statements are fully distributed. Also, our objective is to directly prune the set of semantics-preserving transformations from the transformations (or, schedules) that do not implement the fusion of statements. Note that checking if given a schedule corresponds to fusing the statements is highly impractical: it implies the need to

enumerate all possible valid schedules and check this property in each case. We define the added affine constraints the schedules must respect in order to implement fusion using the following *fusibility* criterion.

Definition 1 (Fusibility of statements): Consider two statements R, S such that R is surrounded by d^R loops and S by d^S loops. They can share p common outer loops if $\forall k \in \{1 \dots p\}$, there exist two semantics-preserving schedules Θ_k^R and Θ_k^S such that:

$$(i) \quad |\Theta_k^R(\vec{0}) - \Theta_k^S(\vec{0})| < c$$

$$(ii) \quad \sum_{i=1}^{d^R} \theta_{k,i}^R > 0, \quad \sum_{i=1}^{d^S} \theta_{k,i}^S > 0$$

Condition (i) ensures that the number of iterations that are peeled from the loops is not greater than c ; it implies that the remaining iterations of R and S will be fused under a common loop.¹ Since schedule coefficients are restricted to be non-negative, c is simply the difference between the constant parametric parts of the schedules. Technically, c is only an estimate of the number of unfused instances, which serves the purpose of this paper. Determining the exact number of fused instances requires one to resort to complex techniques such as counting the number of points in parametric polyhedra [13]. Condition (ii) ensures that the schedule row k has non-null values for the coefficients attached to the loop iterators, that is, (ii) ensures that Θ_k^R and Θ_k^S are not constant schedules. This condition is required to guarantee that Θ_k^R and Θ_k^S represent an interleaving of statement instances in the target code, and not simply a case of statement interleaving.

The only restriction on the Θ coefficients is $\theta_{i,j} \in \mathbb{N}$. Thus this definition takes into account any composition of loop interchange, skewing, multidimensional shifting, peeling and distribution that is required to fuse the statements under a common outer loop, possibly with prolog and epilog.

To ensure that two statements are fusible, we can build a Parametric Integer Program [7] with sufficient constraints for the existence of a semantics-preserving multidimensional schedule [14], [12], in conjunction with the constraints imposed by Definition 1. If this PIP has a solution, then the two statements are fusible.

B. Modeling program partitioning

Our objective is to model the search space that contains all possible partitions of a program, such that statements in the same class can be fused under a common outer loop. In addition, we also require the class identifier to reflect the order in which classes are executed, to model code motion. A general framework for this purpose has been developed by Pouchet [12] in the context of multi-level partitions using arbitrary scheduling coefficients. However in the context of the present work we limit ourselves to the modeling of fusible

¹To ensure that this statement holds true in all cases, further constraints to preprocess the iteration domains are needed [12] They are omitted here for the sake of clarity and do not impact the applicability of this definition.

statements at the outer loop level only, a restriction of the general case.

Returning to the 2mm example of Figure 3. The partitioning is $p_{opteron} = \{\{R\}, \{T, S\}, \{U\}\}$. We represent this partitioning using a vector representing the *statement interleaving* at the outer-most loop level. To reason about it, one may associate an identifier id^S to each statement S such that their ordering encodes exactly the ordering and fusion information for the outer loop level. Using this notation, one gets for $p_{opteron}$ that $\{id^R = 0, id^S = 1, id^T = 1, id^U = 2\}$. This is noted $\vec{f} = (0 \ 1 \ 1 \ 2)$.

We explicitly make \vec{f} exhibit important structural properties of the transformed loop nest:

- 1) if $f_i = f_j$ then the statements i and j share (at least) 1 common loop;
- 2) if $f_i < f_j$ then the statements i and j do not share any common loop, and i is executed before j .

However, intuitively, several choices for \vec{f} represent the same statement interleaving: for example, the transformed code is invariant to translation of all coefficients, or by multiplication of all coefficients by a non-negative constant. Consider the following example, for three statements R, S and T :

$$\vec{f} = (0 \ 2 \ 2)$$

This ordering defines that S and T are fused together, and that R is not and is executed before S and T . An equivalent description is:

$$\vec{f}t = (0 \ 1 \ 1)$$

We observe that the number of duplicates using such a representation grows exponentially with the size of \vec{f} . As a major concern is to prevent duplicates in the space, we use an internal encoding for this problem such that the resulting space contains one and exactly one point per distinct partitioning [12].

The search space is modeled as a convex set of candidate partitions, defined with affine inequalities. There are several motivating factors. The set of possible partitions of a program is extremely large (on the order of 10^{12} possibilities for 14 elements [15], with a super-exponential growth), while the space complexity of our convex set hardly depends on the cardinality of the set. Also, removing a subset of unwanted partitions is made tractable as it involves adding affine constraint(s) to the space, in contrast to other representations that would require enumerating all elements for elimination. Finally, the issue of efficiently scanning a search space represented as a well-formed polytope has been addressed [16], [2], and these techniques apply directly.

C. Pruning for semantics preservation

Previous research on building a convex search space of legal affine schedules highlighted the benefits of integrating the legality criterion directly into the search space, leading to orders of magnitude smaller search spaces [16], [2]. This is critical to allow any iterative search method to focus on relevant candidates only.

To remove all non-valid partitions, we prune the space of all partitions that do not verify Definition 1 for the statements belonging to the same class. Technically, we use an algorithm which iterates on possible partitions and eliminates all invalid candidates based on a graph representation of the problem [12]. To improve the speed of this algorithm, we also leverage some critical properties of fusibility. The algorithm iterates on possible partitions starting from the smallest size for the classes of the partition, leveraging that a superset of an infusible set is not fusible. We also leverage a reduction of the problem of the transitivity of fusibility to the computation of the existence of pairwise compatible loop permutations, as defined in [12]. In practice this algorithm proved to be very fast, and for instance computing all semantics-preserving interleavings at the first dimension takes less than 0.5 second for the benchmark ludcmp, pruning the set from about 10^{12} possible partitions to the remaining 8 valid ones.

V. MODEL-DRIVEN OPTIMIZATIONS

Given a partitioning of the program statements, the second step is to perform aggressive, model-driven optimizations that respect this partitioning. We first discuss the computation of a sequence of loop transformations that implements the partitioning, and produce tiled parallel code when possible in Section V-A. We then present in Section V-B our approach for model-driven vectorization in the polyhedral model.

A. Tiling hyperplanes

The first model-driven optimization we consider applies, individually on each class of the partition, a polyhedral transformation which implements the interleaving of statements via a possibly complex composition of multi-dimensional tiling, fusion, skewing, interchange, shifting, and peeling. It is known as the Tiling Hyperplanes method [10], [3], and we *restrict it to operate locally on each class of the partition*.

The tiling hyperplane method has proved to be very effective in integrating loop tiling into polyhedral transformation sequences [17], [18], [10]. Bondhugula et al. proposed an integrated optimization scheme that seeks to maximally fuse a group of statements, while making the outer loops permutable [10], [3]. A schedule is computed such that parallel loops are brought to the outer levels, if possible. This technique is applied locally on each class, thereby maximizing parallelism at the level of that class, without disturbing the outer level partitioning.

1) *Legality of tiling*: Tiling a loop nest is legal if the loops to be tiled can be permuted [17]. If the loops are indeed permutable, then there would be no dependence path going in and then out of a given tile. This is also known as the Forward Communication Only [19] property, and can be encoded as an affine scheduling constraint [17], [10].

Definition 2 (Legality of Tiling): Given two statements R and S . Tiling is legal if, for all dimensions d to be tiled and for all dependences $\mathcal{D}_{R,S}$:

$$\Theta_d^S(\vec{x}_S) - \Theta_d^R(\vec{x}_R) \geq 0, \quad \langle x_R, x_S \rangle \in \mathcal{D}_{R,S} \quad (1)$$

Definition 2 ensures that *none* of the dependences point backward along any of the dimensions to be tiled. This is a stricter condition than simple semantics-preservation.

Schedules are selected such that each dimension is independent with respect to all others: this leads to a one-to-one mapping. Rectangular or nearly rectangular blocks are achieved when possible, avoiding complex loop bounds required for arbitrarily shaped tiles.

The algorithm proceeds by computing the schedule level by level, from the outermost to the innermost. At each level, a set of legal hyperplanes is computed for the statements, according to the cost model defined in Section V-A2. Dependences satisfied by these hyperplanes are marked, and another set is computed for the next level such that the new set is independent with respect to all previously computed sets, and so on until all dependences have been marked satisfied. At a given loop level, if it is not possible to find legal hyperplanes for all statements, the statements are split [10], resulting in a loop distribution at the level. We note that from the approach to construction of valid partitions, this cannot occur at the outermost loop level.

2) *Static cost model*: Infinitely many schedules satisfy the criterion (1) for legality of tiling. As a second objective, to achieve good temporal locality we seek a schedule that minimizes the hyperplane distance between dependent iterations [10]. For code with affine dependences, the distance between dependent iterations can always be bounded by an affine function of the global parameters \vec{n} .

$$\mathbf{u} \cdot \vec{n} + w \geq \Theta_d^S(\vec{x}_S) - \Theta_d^R(\vec{x}_R) \quad \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S} \quad (2)$$

$$\mathbf{u} \in \mathbb{N}^p, w \in \mathbb{N}$$

The form $\mathbf{u} \cdot \vec{n} + w$ is an upper bound on the distance between all dependent iterations, and thus directly impacts single-thread locality as well as communication volume in the context of parallelization. It is thus desirable to seek transformations that minimize it. The legality and bounding function constraints from (1) and (2) are formulated as a single Integer Linear Program. The coefficients of Θ_d and those of the bounding function, i.e., \mathbf{u} , w , are the only unknowns left. A solution is found to minimize the value of \mathbf{u} and w to obtain the unknown coefficients of Θ_d .

$$\text{minimize}_{\prec} (\mathbf{u}, w, \dots, \theta_{d,i}, \dots) \quad (3)$$

For each class of the partition (i.e., each group of fused statements), several goals are achieved through this cost model: maximizing coarse-grained parallelism, minimizing communication and frequency of synchronization, and maximizing locality [10]. Since outer permutable bands are exposed, multidimensional tiling can be applied on them. Tiles can be executed in parallel or with a pipeline-parallel schedule. In the generated programs, parallelization is obtained by marking transformed parallel loops with OpenMP pragmas.

3) *Profitability of the transformation*: The profitability of the Tiling Hyperplane method is complex to assess in its general formulation, as it is characterized by the profitability

of loop fusion and the impact on subsequent vectorization. Our technique has removed the profitability estimate of outer-loop fusion/distribution, since we empirically evaluate all possible choices. In addition, we rely on a second stage dedicated to expose inner loops which are good vectorization candidates. So the problem of the profitability of the Tiling Hyperplane is reduced to the effectiveness of maximizing data locality in a given class, while outer-parallelism and vectorizable loops are made independent to the problem. Technically, one should consider the profitability of multi-level statement interleaving to guarantee that each possible loop structure is evaluated in order to find the best one, trading parallelism and locality at each loop level. However, focusing only on the outer level carries the most important changes in parallelism and communication possibilities. In our optimization algorithm, *we chose to systematically apply the tiling hyperplane method on each class of the partition*.

4) *Applying tiling on a transformed loop nest*: Tiling a permutable loop nest is profitable in particular when there is reuse of data elements within the execution of a tile. Another criterion to take into account is to preserve enough iterations at the inner-most loop level to allow for a profitable steady-state for vector operations within the execution of a tile. Our extremely simple algorithm to determine the tiling of a loop nest proceeds as follows:

- 1) compute the order of magnitude of data reuse in the loop nest;
- 2) compute the depth of the loop nest;
- 3) if there is $O(N)$ reuse within a loop, and the loop nest depth is greater than 1 then tile the loop nest.

To achieve maximal performance it is expected that tuning the tile sizes can provide improvement, however the problem of computing the best tile sizes for a loop nest is beyond the scope of this paper. In our experiments tile sizes are computed such that data accessed by each tile roughly fits in the L1 cache.

B. Vectorization

On modern, SIMD-capable architectures, vectorization is a key for performance. The acceleration factor is a conjunction of several elements including, but not limited to, the number of elements packed in a vector and the throughput of the vector units. Making the most of SIMD units requires to operate with the two following categories of optimizations:

- 1) high-level transformations, to expose parallel inner loops with maximally-aligned, minimally-strided accesses, which are good candidates for vectorization;
- 2) low-level transformations, to deal with hardware constraints such as realignment, vector packing or vector instruction selection.

In our framework we rely on the back-end compiler to produce *vectorized* code. A major challenge for production compilers is to detect and possibly transform the code to expose good vectorizable loops. They are geared towards the common case, and must provide extremely fast compilation time. They lack the precision and expressiveness of

	description	#loops	#stmts	#refs	#deps	#part.	#valid	Variability	Pb. Size
2mm	Linear algebra (BLAS3)	6	4	8	12	75	12	✓	1024x1024
3mm	Linear algebra (BLAS3)	9	6	12	19	4683	128	✓	1024x1024
adi	Stencil (2D)	11	8	36	188	545835	1		1024x1024
atax	Linear algebra (BLAS2)	4	4	10	12	75	16	✓	8000x8000
bicg	Linear algebra (BLAS2)	3	4	10	10	75	26	✓	8000x8000
correl	Correlation (PCA: StatLib)	5	6	12	14	4683	176	✓	500x500
covar	Covariance (PCA: StatLib)	7	7	13	26	47293	96	✓	500x500
doitgen	Linear algebra	5	3	7	8	13	4		128x128x128
gemm	Linear algebra (BLAS3)	3	2	6	6	3	2		1024x1024
gemver	Linear algebra (BLAS2)	7	4	19	13	75	8	✓	8000x8000
gesummv	Linear algebra (BLAS2)	2	5	15	17	541	44	✓	8000x8000
gramschmidt	Matrix normalization	6	7	17	34	47293	1		512x512
jacobi-2d	Stencil (2D)	5	2	8	14	3	1		20x1024x1024
lu	Matrix decomposition	4	2	7	10	3	1		1024x1024
ludcmp	Solver	9	15	40	188	10 ¹²	20	✓	1024x1024
seidel	Stencil (2D)	3	1	10	27	1	1		20x1024x1024

Fig. 6. Summary of the optimization process

the polyhedral framework, implementing instead approximate techniques. A consequence is that such compilers often fail to compute a restructuring loop transformation to expose the best candidate for the inner-most loops; or may even fail to detect vectorizable loops because of the complexity of the surrounding tile loop bounds. Moreover, their general-purpose heuristics may produce unvectorized code when dealing with complex parametric loop bounds or array access alignments.

Our approach to vectorization leverages recent analytical modeling results by Trifunovic et al. [20]. We take advantage of the polyhedral representation to aggressively restructure the code to expose vectorizable inner loops. Cost analysis and loop transformations are performed in a very expressive framework, while deferring to the back-end compiler the task of performing low-level transformations. In addition we bypass the compiler’s high-level vectorization analysis stage by marking loops with `#pragma vector always` and `#pragma ivdep`, when applicable.

1) *The Algorithm*: Our algorithm proceeds level-by-level, from the outer-most loop level to the inner-most. For each loop at that level, candidates for vectorization are computed such that: (1) the loop can be moved to the inner-most position — via a sequence of loop interchanges — while preserving the semantics; and (2) moving this loop to the inner-most position does not remove thread-level parallelism. For each loop in the set of candidates we compute a cost metric based on the maximal distance (in memory) between data elements accessed by two consecutive iterations of this loop [20], considering all statements enclosed in this loop. The algorithm then moves to the next loop level, until all candidate loops for vectorization have been annotated with the cost metric. Loops with the best metric are then sunk inwards to the inner-most position, with a sequence of permutations captured within the polyhedral representation. The tiling hyperplane method guarantees that this sinking operation is always legal: this seamless coordination of the two methods is a key benefit of a polyhedral compilation framework. Note that because of parametric and possibly non-matching loop bounds, this transformation may result in additional prolog/epilog code surrounding the loops. This is handled seamlessly in the polyhedral representation but would have posed a major challenge

to standard transformation frameworks.

2) *Profitability*: Combining the approach of Trifunovic et al. with the tiling hyperplane method leads to a very robust algorithm: it identifies the most profitable vectorization alternative in most cases. Because we do not alter the general code structure (no subsequent distribution or parallelism removal), the profitability is only connected to sinking inwards a parallel loop that accesses data in a more contiguous fashion. When our algorithm fails at exposing the most profitable inner-most loop, the chosen loop is extremely likely to be a better candidate for vectorization than the one it replaces.

VI. EXPERIMENTAL RESULTS

The automatic optimization and parallelization framework has been implemented in PoCC, the *Polyhedral Compiler Collection*,² a complete source-to-source polyhedral compiler integrating well established free software for polyhedral optimization.

A. Experimental setup

We performed our experiments on two modern multi-core machines: a 4-socket Intel hex-core Xeon E7450 (Dunnington) running at 2.4GHz with 64GB of memory (24 cores, 24 hardware threads) and a 4-socket AMD quad-core Opteron 8380 (Shanghai) running at 2.50GHz (16 cores, 16 hardware threads) with 64GB of memory. We also experimented on a representative of low-cost computing platforms, an Atom 230 processor running at 1.6GHz with 1GB of memory (1 core, 2 hardware threads). All systems ran Linux 2.6.x. We used ICC 11.1 for the Xeon and Opteron machines, and GCC 4.3.3 for the Atom. The compilation flags used for the original code were the same as for the different tested versions; they are reported in Figure 7.

B. Summary of experiments

Figure 6 presents the main characteristics of our benchmark suite. We considered 16 benchmarks from the PolyBench test suite [21]. For most programs, the execution time of the original code using the specified problem sizes is below 3 seconds. In Figure 6, we report for each benchmark some

²PoCC is available at <http://pocc.sourceforge.net>

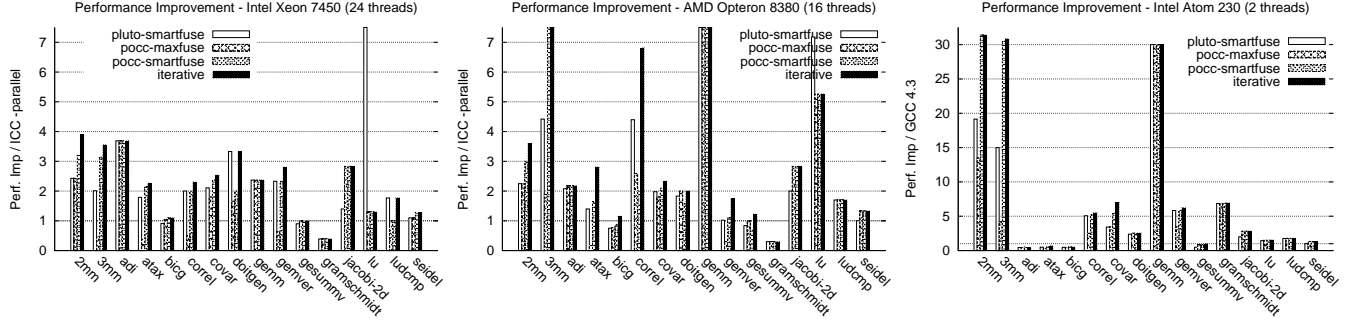


Fig. 7. Performance improvement of (a) state-of-the-art smart fusion without dedicated vectorization stage (pluto-smartfuse); (b) two specific partitionings of our search space: maximal fusion and smart fusion, both with dedicated vectorization stage (pocc-maxfuse and pocc-smartfuse); and (c) the best found partitioning after empirical search (iterative). Baseline is ICC -fast -parallel -openmp for Xeon and Opteron, GCC -O3 -fopenmp -mssse3 -march=prescott -mtune=pentium -funroll-loops for Atom.

information on the considered SCoP: (#loops the number of loops, #stmts the number of statements, #refs the number of array references, #deps the number of dependence polyhedra). We report also the number of possible (including invalid) partitions as #part., and the number of semantics-preserving partitions #valid to highlight the pruning factor enabled by our algorithm. We also check the column Variability each time we observed a 5% or more difference between the best versions found for a platform and its execution on the other platforms, this to emphasize the requirement for a tuning of the partitioning selection. Finally, we also report the dataset size used for the benchmarks (Pb. Size).

C. Detailed performance evaluation

The time to compute the space, pick a candidate and compute a full transformation is negligible with respect to the compilation and execution time of the tested versions. In our experiments, the full optimization process for the 16 presented benchmarks took less than one hour on the Atom, the slowest machine. This time is totally dominated by the execution time of each candidate; had we used a smaller/larger dataset sizes the optimization time would have decreased/increased.

1) *Performance improvement*: We report in Figure 7 the performance improvement of our technique when compared to the native production compiler with aggressive optimization flags enabled used as the baseline (performance improvement = 1). To study in a fair fashion the benefit of our methods, we particularly look at two specific partitionings:

- *maxfuse*, which corresponds to applying the tiling hyper-plane method on the full program instead of locally to each class of the partition;
- *smartfuse*, which corresponds to a partitioning where statements that do not share any data reuse are put in different classes. This is considered the state-of-the-art [3].

To further emphasize the benefits of our approach, we report the performance improvement of smart fusion when used without our complementary step for vectorization (pluto-smartfuse), and with it (pocc-smartfuse). The best performance improvement found by our combined approach is reported in iterative.

We obtain significant performance improvements over the native compiler, above $2\times$ better on average for the Xeon and $2.5\times$ better for the Opteron. For most programs, ICC was able to automatically parallelize the original code. Still, we exhibit strong improvements, particularly on compute-intensive kernels such as 3mm or correl, up to $8.5\times$ improvement. For gramschmidt with ICC (for both Xeon and Opteron), our polyhedral framework results in decreased performance. The application of tiling increased the complexity of loops and prevented ICC from performing the same scalar optimizations done on the original code.

For the case of Atom, we observed that GCC fails in many situations to utilize both coarse-grain and fine-grain parallelism in the input code. This leads to very large improvements by our framework: up to $30\times$ for matrix multiplications. However for the most memory-bound benchmarks such as atax or bicg, our transformations decreased the performance by a small factor. There was no benefit in introducing more complex control to expose parallelism, as the tested Atom has only a single physical core.

For most benchmarks, smart fusion performs better than maximal fusion, showing the importance of controlling the cache pressure and exposing enough inner-parallel loops. A model-driven approach such as *maxfuse* which looks for the minimization of synchronizations and maximization of locality is still likely to provide a performance improvement over general-purpose heuristics implemented in a production compiler, as shown in Figure 7. Such examples are shown for the benchmarks with only one possible partitioning, thus equivalent to applying maxfuse to the full program. However, empirical search is needed for 9 out of 16 benchmarks to obtain the best performance, for a benefit of up to $2\times$ over smart fusion.

2) *Performance portability*: The optimal partitioning depends on the program, but is also influenced by the target machine. This is shown by the Variability column of Table 6. For 9 of the 11 benchmarks with more than one legal partitioning, there exists no partitioning such that when it is executed on all three machines, it performs within 5% of the optimal one found for each machine. Increasing the threshold to 10%,

this is still the case for 7 of the 11 benchmarks.

The trade-off between coarse-grain parallelization, locality and vectorization is very difficult to capture. Using our framework, tuning the trade-off between fusion and distribution drives the effectiveness of subsequent well-defined cost models used to transform the code to expose different choices of locality and parallelization. Our iterative technique automatically discovers the partitioning with optimal performance, whatever the specifics of the program, compiler and architecture.

VII. RELATED WORK

Iterative optimization has proved its effectiveness in providing performance improvements over a broad range of architectures and compilation scenarios [22], [23], [24], [25], [26], [27], [2], [28]. However, none of the previous approaches attempt to construct program transformation sequences as complex and as extensive as the ones presented in this paper while pruning the search space to semantics-preserving candidates only.

Loop fusion heuristics were initially designed as locality-enhancing optimizations, in isolation from other loop nest transformations [4], [29], [30], [31]. These non-polyhedral approach are restricted in their ability to find complex partitions, or model the interplay of loop fusion with equally important optimizations such as loop tiling. The lack of a powerful representation for dependences and composition of transformations also restricted the study of enabling loop transformations to enhance the applicability of loop fusion.

Several heuristics for loop fusion combined with tiling have been proposed [32], [26], but do not capture the interplay between loop transformations, back-end optimizations performed by the compiler, and components of the target architecture. Megiddo and Sarkar [30] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed. Decoupling parallelization and fusion can miss interesting solutions that would have been identified if the set of legal fusion choices were directly cast into the framework.

Darte et al. [33], [34], [35] studied fusion for data-parallelization, but only in combination with shifting. These important complexity results have been influential in our successful selection of a hybrid optimization scheme, focusing the iterative search on the most combinatorially explosive optimization — loop fusion — while designing a heuristic and an analytical profitability model for the other affine transformations enabling loop tiling and data parallelization.

Recent research on integrating fusion and tiling in a single heuristic based on the polyhedral model led to the Pluto framework by Bondhugula et al. [10], [3]. It inherits the flexibility of the tiling hyperplane method [17], [36] to build complex sequences of enabling and communication-minimizing transformations, subsuming most compositions of loop transformations into a single optimization step. It does identify excellent parallelism-locality trade-offs using a target-independent cost model. However as shown in this paper, better solutions can be found via empirical search.

Powerful semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [37], [38], [1], [39], [40]. They capture partitioning, but neither do they define automatic iteration schemes nor do they integrate a model-based heuristic to construct profitable parallelization and tiling strategies. The polyhedral model creates many more opportunities for the construction of loop nest optimizers and parallelizing compilers. It is currently being integrated in production compilers, including GCC 4.5 and the IBM XL compiler.

VIII. CONCLUSION

This paper addressed the problem of optimizing and parallelizing programs automatically, focusing on static control loop nests. Our approach departs from the traditional best-effort compiler optimizations, aiming for performance portability across a variety of shared-memory multiprocessors. We proposed a combined iterative and model-driven approach, leveraging a state-of-the-art parallelization method based on loop tiling, and combining it with a novel feedback-directed scheme for loop fusion and distribution.

Our technique builds an expressive search space of loop transformation sequences, expressed in the polyhedral model as a set of affine scheduling functions. The search space encompasses complex compositions of loop transformations, including loop fusion and distribution, loop tiling for parallelism and locality (caches, registers), loop interchange, and loop shifting (pipelining). We proposed a convex encoding of all legal transformed program versions as the space to search.

We performed experiments on three different platforms: a 24-core Xeon, a 16-core Opteron, and a single-core low-power Atom processor. Our experiments confirm that no single program version performs equally well on different targets, with penalties reaching $2\times$ when running the best version for a given target on a different target. We also consistently demonstrate strong performance improvements over the state-of-the-art model-based compilers, with performance improvement factors up to $8\times$ over Intel's compiler. In the future, we plan to study the applicability of machine learning techniques to prune our hybrid optimization space or predict the performance of transformed program versions. We will also continue to look for ways of building an even more expressive space, and narrowing down the gap with respect to peak performance on a wide set of benchmarks and target architectures.

ACKNOWLEDGMENT

This work was supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, the U.S. National Science Foundation through awards 0926687/0926688, and by the U.S. Army through contract W911NF-10-1-0004. It was also partly supported by the European Commission through the FP6 project SARC id. 027648.

REFERENCES

- [1] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parelo, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Intl. J. of Parallel Programming*, vol. 34, no. 3, 2006.
- [2] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08)*. ACM Press, 2008, pp. 90–100.
- [3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2008.
- [4] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing*, 1993, pp. 301–320.
- [5] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.
- [6] M. Wolfe, *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.
- [7] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.
- [8] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, ser. LNCS, Paphos, Cyprus, Mar. 2010, pp. 283–303.
- [9] P. Feautrier, "Some efficient solutions to the affine scheduling problem, part II: multidimensional time," *Intl. J. of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992.
- [10] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International conference on Compiler Construction (ETAPS CC)*, Apr. 2008.
- [11] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, Juan-les-Pins, France, Sep. 2004, pp. 7–16.
- [12] L.-N. Pouchet, "Iterative optimization in the polyhedral model," Ph.D. dissertation, INRIA Saclay and University of Paris-Sud 11, Jan. 2010.
- [13] P. Clauss, "Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs," in *Intl. Conf. on Supercomputing*, Philadelphia, May 1996, pp. 278–285.
- [14] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, pp. 389–420, 1992.
- [15] N. J. A. Sloane, "Sequence a000670," The On-Line Encyclopedia of Integer Sequences, <http://www.research.att.com/~njas/sequences/A000670>.
- [16] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative optimization in the polyhedral model: Part I, one-dimensional time," in *Proc. of the IEEE/ACM Fifth Intl. Symp. on Code Generation and Optimization (CGO'07)*. IEEE Comp. Soc. press, 2007, pp. 144–156.
- [17] F. Irigoin and R. Triolet, "Supernode partitioning," in *ACM SIGPLAN Principles of Programming Languages*, 1988, pp. 319–329.
- [18] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 108–230, 1992.
- [19] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Fakultät für mathematik und informatik, universität Passau," 2004.
- [20] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 327–337.
- [21] "PolyBenchs 1.0," available at <http://www-rocq.inria.fr/pouchet/software/polybenchs>.
- [22] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," in *W. on Profile and Feedback Directed Compilation*, Paris, Oct. 1998.
- [23] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning," *SIGPLAN Not.*, vol. 38, no. 5, pp. 77–90, 2003.
- [24] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *Proc. of the Intl. Symposium on Code Generation and Optimization (CGO'06)*, Washington, 2006, pp. 295–305.
- [25] S. Long and G. Fursin, "A heuristic search algorithm based on unified transformation framework," in *Proc. of the 2005 Intl. Conf. on Parallel Processing Workshops (ICPPW'05)*. Washington, DC, USA: IEEE Comp. Soc., 2005, pp. 137–144.
- [26] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *Proc. of the 20th Intl. Conf. on Supercomputing (ICS'06)*. ACM press, 2006, pp. 249–258.
- [27] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," in *Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation (PLDI'05)*. ACM, 2005, pp. 315–326.
- [28] Y. Voronenko, F. de Mesmay, and M. Püschel, "Computer generation of general size linear transform libraries," in *Intl. Symp. on Code Generation and Optimization (CGO'09)*, Mar. 2009.
- [29] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.
- [30] N. Megiddo and V. Sarkar, "Optimal weighted loop fusion for parallel programs," in *symposium on Parallel Algorithms and Architectures*, 1997, pp. 282–291.
- [31] S. Singhai and K. McKinley, "A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality," *The Computer Journal*, vol. 40, no. 6, pp. 340–355, 1997.
- [32] M. Wolf, D. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996, pp. 274–286.
- [33] A. Darte, G.-A. Silber, and F. Vivien, "Combining retiming and scheduling techniques for loop parallelization and loop tiling," *Parallel Proc. Letters*, vol. 7, no. 4, pp. 379–392, 1997.
- [34] A. Darte, "On the complexity of loop fusion," *Parallel Computing*, pp. 149–157, 1999.
- [35] A. Darte and G. Huard, "Loop shifting for loop parallelization," ENS Lyon, Tech. Rep. RR2000-22, May 2000.
- [36] M. Griebl, P. Faber, and C. Lengauer, "Space-time mapping and tiling – a helpful combination," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 3, pp. 221–246, Mar. 2004.
- [37] W. Kelly, "Optimization within a unified transformation framework," Department of Computer Science, University of Maryland at College Park, Tech. Rep. CS-TR-3725, 1996.
- [38] A. Cohen, S. Girbal, D. Parelo, M. Sigler, O. Temam, and N. Vasilache, "Facilitating the search for compositions of program transformations," in *ACM International conference on Supercomputing*, Jun. 2005, pp. 151–160.
- [39] C. Chen, J. Chame, and M. Hall, "CHILL: A framework for composing high-level loop transformations," U. of Southern California, Tech. Rep. 08-897, 2008.
- [40] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable autotuning framework for computer optimization," in *IPDPS'09*, Rome, May 2009.